# JAVA APPLICATION PROGRAMMER'S INTERFACE (API) FOR HEURISTIC TREE SEARCH ALGORITHMS

by

Patrick J. Clemins, B.S.

A Thesis submitted to the Faculty
of the Graduate School,
Marquette University,
in Partial Fulfillment of
the Requirements for
the Degree of
Master of Science

Milwaukee, Wisconsin
December, 2000

# Preface

Artificial intelligence (AI) has been an ongoing research area since the advent of computers. There are many different aspects to AI including speech synthesis, image recognition, and problem solving. There are also many different algorithms that can be used to solve these problems. Some are complex and attempt to use AI techniques to solve the problem. Others search all possible states until the solution is reached. The set of all possible configurations of the problem is called the solution space. In order to search the solution space in an organized fashion, it must be represented by a data structure. A tree is one common way to represent the solution space of a problem. There are many possible ways to search a tree. Computer algorithms to search trees have been developed, refined and optimized by many researchers and much has been published on this topic.

With the advent of object-oriented design and the development of object-oriented languages, new programming techniques have emerged. One of the big advantages these new programming languages promote is reusable code. Although the aforementioned topics, object-oriented design and tree searching, have received much attention separately, there has been little effort to use these techniques together. Previously, a tree based AI problem was solved by selecting an algorithm and then tweaking the algorithm to fit the problem at hand. Then, when another problem was presented, the algorithm would again have to be developed from the beginning since the previous algorithm was customized around the specific problem. This is a large waste of time and increases the amount of new code that must be developed. This thesis asks the question, "is it possible to code the algorithm once, in a generalized way, using object oriented techniques, so that the same code could be used again for another problem."

This question is answered in the affirmative by describing an application programmer's interface (API) which can be used to solve any tree based AI problem without having to write or customize any algorithm code. The API contains generalized versions of four popular tree search algorithms. The only code required of the programmer that wishes to use this API is two or three functions which describe the characteristics of the problem. This API can speed development of solutions to tree-based problems and can be used to quickly compare the effectiveness of different algorithms in different circumstances.

## Author's Note

I first heard about object oriented programming in EECE 191 – Introduction to Computer Hardware and Software at Marquette University. That semester, the class was taught by Prof. Susan Reidel, and our project was an elevator control simulation. Since then, I have always tried to use that technology in all of my programming classes. So, when I took Algorithm Analysis, taught by Dr. Xin Feng, and learned about different heuristic tree search algorithms, I wondered how I could apply object oriented technology to those algorithms. For the class term project, I wrote an A* algorithm in C to solve the 8-puzzle. I never quite got that to work, but I became interested in writing the algorithm in a language I was more familiar with.

When it became time to decide what to do for my master's research, I remembered that project, and thought about how I could extend it into a master's thesis. I decided that I would make my implementation general purpose so I would not have to write the algorithm code again. After a little research, I realized that such an undertaking had not been done. Then, I thought that I might as well include a few more algorithms and create some demonstration problems. After I presented the intended design of the API at the Midwest Computing Conference, I realized that there was an interest in such a framework. Some of the design has changed since that presentation, but the intent is the same, to make life easier for programmers.

# Acknowledgements

I would like to thank my family for their patience and understanding in my career goal change from industry to academia.  They have been always been supportive of my endeavors, no matter how much they may not understand them.

I would also like to thank my Marquette family, the men of Triangle Fraternity, for their support and letting me know that there is more to a college experience than simply the development of the mind.  There is also development of the soul.

Thank you to the faculty of the Computer Science and Electrical and Computer Engineering departments.  Not only are they wonderful educators, but wonderful people as well.  Each seems truly interested in my success as a student and researcher.

Finally, thanks to all whom I made read this paper against their will so that they could provide feedback about grammar and content.  Your help is greatly appreciated.

# Table of Contents

Background ....................................................................................................1

Current Implementations ....................................................................................................5

The API Exposed and Explained ....................................................................................................7

- Solvable Interface ....................................................................................................7
- Hsolvable Interface (extends Solvable) ....................................................................................................7
- DepthFirstSearch and BreadthFirstSearch Classes ....................................................................................................8
- BestFirstSearch Class ....................................................................................................8
- AStarSearch Class ....................................................................................................9
- ValuedNode Class (implements Comparable) ....................................................................................................9
- Tree Class ....................................................................................................10

Implementation ....................................................................................................12

- Solvable Interface ....................................................................................................12
- HSolvable Interface ....................................................................................................12
- TreeSearch Class ....................................................................................................12
- DepthFirstSearch Class ....................................................................................................15
- BreadthFirstSearch Class ....................................................................................................16
- HTreeSearch Class ....................................................................................................16
- BestFirstSearch Class ....................................................................................................17
- AStarSearch Class ....................................................................................................18
- ValuedNode Class ....................................................................................................19
- Tree Class ....................................................................................................20

Use of the API ....................................................................................................29

- 8-Puzzle ....................................................................................................29
- Maze Puzzle ....................................................................................................37
- Other Applications ....................................................................................................41

Conclusion ....................................................................................................43

References ....................................................................................................45

Bibliography ....................................................................................................45

Appendix A – Source Code ....................................................................................................46

# Table of Figures

Figure 1: A Simple Tree....1
Figure 2: Depth-first Search....2
Figure 3: Breadth-first Search....2
Figure 4: Best-first Search....3
Figure 5: A* Search....4
Figure 6: 8-Puzzle....29
Figure 7: Ex. 1 - Start....36
Figure 8: Ex. 1 - Goal....36
Figure 9: Ex. 2 - Start....36
Figure 10: Ex. 2 - Goal....36
Figure 11: Maze....37
Figure 12: Depth-first Search....41
Figure 13: Breadth-first Search....41
Figure 14: Best-first Search....41
Figure 15: A* Search....41

## Background

A tree (Figure 1) is a hierarchical data structure. Each circle represents one state of the problem and is referred to as a node. The top node, the one from which all others are descended, is called the root (node 1). The children of a node are all nodes that descend from it. The parent of a node is the node from which it is descended. For example, node 2 is a child of node 1 and node 1 is the parent of node 2. A leaf is any node without any children (node 2).

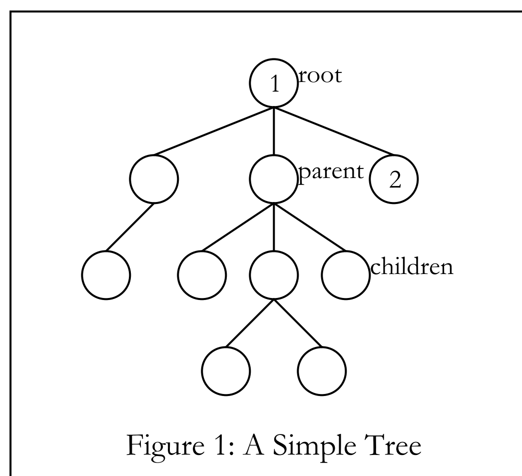


Figure 1: A Simple Tree

There are two different strategies for searching a solution space. One of these strategies is data-driven. A data-driven strategy starts at the root and traverses the branches of the tree until it finds the solution. The other is a goal-driven strategy that starts at the solution and then tries to find a path back to the starting point. The goal-driven strategy is also commonly called backtracking. All four algorithms considered here are data-driven strategies.

There are four different search algorithms that will be implemented in this application programmer's interface (API). These four algorithms are depth-first, breadth-first, best-first and the A* (A-star) algorithm. The A* algorithm is actually a best-first algorithm that tends to favor shorter solution paths. These algorithms were chosen because they are the most commonly used tree search algorithms in current literature. The four algorithms are explained below.

The depth-first search starts by expanding the root to see if the solution is generated. If none of the newly generated children are the solution, the search continues by expanding one of the new children. This continues until a leaf is encountered. A leaf is a node without any children. After a leaf is encountered, the search goes back to the closest node that has not yet been expanded. This continues until all nodes are expanded or the solution is found. Figure 2 shows how a depth-first search would progress through a sample tree. The numbers shown are the sequence in which the nodes are expanded.

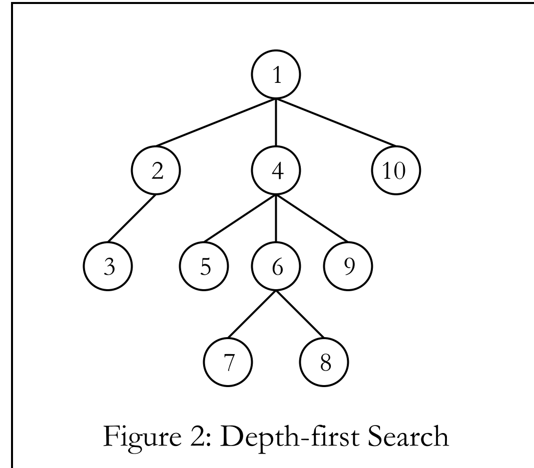

Figure 2: Depth-first Search

The breadth-first search starts by expanding the root to see if the solution is generated. If not, the search continues to expand each child of the root until all of the children are expanded. After all of the children are expanded, the search expands each of the root's children. This continues until the solution is found or all nodes are searched. Figure 3 shows how a breadth-first search would progress through a sample tree.

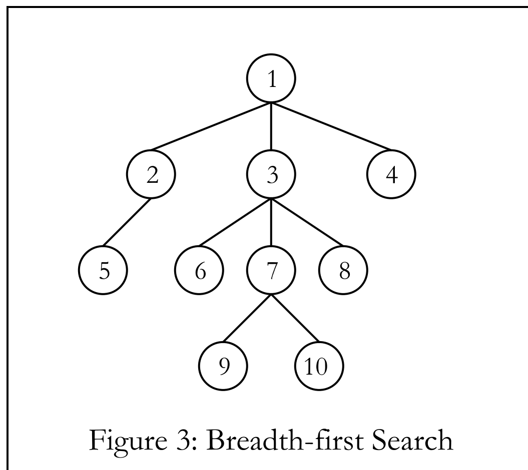

Figure 3: Breadth-first Search

The two searching algorithms discussed thus far use no intelligence to determine the best path to the solution. Instead, they systematically search the entire tree. For this reason, these algorithms are sometimes called brute-force algorithms. These algorithms are simple to implement. However, they typically perform a lot of comparisons before they find the

solution. In contrast, the next two algorithms discussed use a heuristic to choose a particular path to follow. A heuristic is a formula used to approximate how close a given state is to the solution. These algorithms maintain a list of open nodes. Open nodes are nodes that have not been expanded yet. The open list is usually ordered based on the heuristic value of each node.

The best-first search starts out by placing the root in the open list. As with the rest of the algorithms, once the solution is found, the search stops. For each iteration of the search, the node with the best ranking (based on the heuristic) is removed from the open list. In this example, the lower the heuristic value, the better the ranking. The highest ranked node's children are examined to see if any of them is the solution. If not, the heuristic function is calculated for each child and the children are added to the open list. No puzzle state may ever be added to the open list twice, so if the child is a duplicate, it is not added to the tree or the open list. Nodes are removed from the open list in this fashion until the list is empty or the solution is found. The search has failed if the open list is empty and the solution has not yet been found. Notice the three nodes with a heuristic value of four near the top of the tree. There is no absolute rule governing which node should be searched first when they both have the same heuristic value, but a generally accepted tiebreaker is whichever node was added to the open list first. Figure 4 shows how a best-first search would traverse a sample tree. The number in the node represents the order in which they were expanded. The bold node is the goal node.

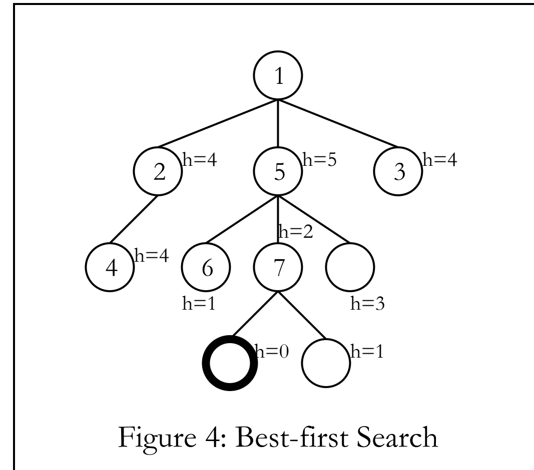

Figure 4: Best-first Search

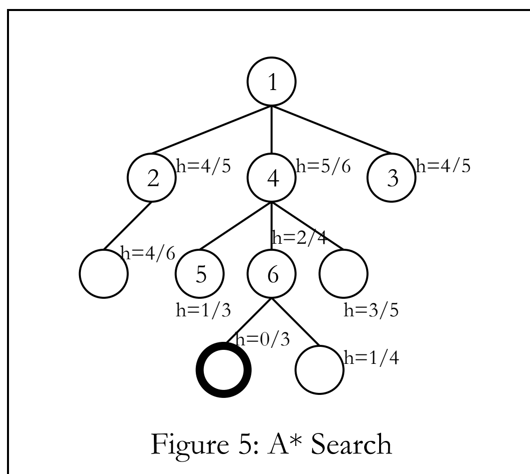

Figure 5: A* Search

The A* algorithm is a modified best-first search. The only change to the best-first search is the way in which the next node is picked from the open list. Instead of picking the next node based solely on the heuristic function, the value used to determine the next node is calculated by adding the heuristic function and the distance from the node to the root. For instance, in the example A* search in Figure 5, the second node would have a value of 5 (4 + 1) while its child would have a value of 6 (4 + 2). Although both nodes return the same value from the heuristic function, node 2 is more desirable because it is higher up in the tree hierarchy. The A* algorithm is modified from the best-first search so that it can find not only the solution, but also the shortest path to the solution. That is why the distance from the root is factored into each node's assigned value. The first number by each node is the heuristic value, and the second is the heuristic value with the depth added to it.

## Current Implementations

The above four algorithms have been implemented numerous times in almost every programming language in existence. Significant research efforts have been focused on efficiently implementing different heuristic algorithms. However, most implementations have been application specific. No one has ever set out to create a common framework in Java that can be used in different applications to ease development. Discussed below are a few examples of what has been done in the past.

Mark Watson has a website dedicated to discussing Java implementations of search algorithms. On his site, he has two applets that show how a depth-first search and a breadth-first search traverse a graph. He talks about creating a Java framework, but his framework only consists of a single abstract search class. This abstract class is then inherited to create the depth-first search and breadth-first search algorithm classes. These two classes he creates from the abstract search class are application specific. He presents some good ideas on using an object-oriented implementation of the algorithms to create a framework, but he does not create one (Watson 1998).

Kim Tracy and Peter Bouthoorn discuss how to implement tree search algorithms in an object-oriented fashion in their book, Object-Oriented: Artificial Intelligence Using C++. They explain and discuss the code for implementing all of the algorithms described above, but each example is written for a specific problem. No mention is made of the idea of creating a generalized framework, but they do provide an interesting discussion on the efficiency of the implementations (Tracy, Bouthoorn, 1997).

The most similar research pertaining to this project is outlined in a paper titled "An Object-Oriented View of Backtracking" by Robert E. Noonan. In this paper, Mr. Noonan presents a reusable backtracking (depth-first search) class. He defines an interface,

Backtracker, which contains five methods: valid(), record(), done(), undo(), and moves(). Mr. Noonan uses a different approach to the depth-first search algorithm. Instead of using a tree data structure, which can take up lots of memory, Mr. Noonan presents a recursive version of the algorithm. Mr. Noonan approaches the framework from an algorithm centric view. Thus, he is only concerned about implementing a framework around a single algorithm. This project, however, takes a data centric view. Instead of only using one algorithm, many algorithms can be developed to search a tree data structure. As a result, this project is more inclusive to various techniques and is more expandable. All problems presented in Mr. Noonan's paper could easily be implemented in this framework (Noonan, 2000).

It should be noted that the goal of this project is not to create a new algorithm or even to improve on an existing algorithm, but to provide easy access to these algorithms. It is intended that people with very little knowledge about how these algorithms work will be able to use them to solve various AI problems.

## The API Exposed and Explained

The API consists of various objects and interfaces to aid in solving problems that can be represented in a tree data structure. The programmer uses these API objects and methods as a basis for writing their own code. The API can be divided into two sections. The algorithm classes and the interfaces are high-level classes designed to be used by an application programmer. The rest of the API consists of support classes. These support classes are used to support the rest of the API and provide programmers with the ability to expand the API by defining different search algorithms. These support objects give a programmer the ability to manipulate a solution space (a tree) and to organize a list of open nodes. Each API object and interface is explained below.

**Solvable Interface**

`Object[ ] generateChildren ()`

Generates an array of possible next states for the object

`boolean equals (Object x)`

Returns true if x is logically equal to the object

This interface consists of two methods, generateChildren() and equals(). These are required so that the algorithm can create the child nodes and determine whether each node meets the solution. Each class that wants to use an algorithm's solve() method must, at a minimum, implement these two methods. If a programmer tries to use one of the algorithms without implementing Solvable, a compile-time error will result.

**Hsolvable Interface (extends Solvable)**

`int fitnessFunction (Object solution)`

Returns an integer specifying how logically close the object is to solution

The HSolvable interface is a sub-interface of Solvable. This means that the Hsolvable interface consists of all of the methods of Solvable and whatever additional

methods it chooses to define. To use a heuristic search algorithm, the two parameters of the solve() method must meet the HSolvable interface. The Hsolvable interface adds the heuristic method, fitnessFunction(), to the original two methods of the Solvable interface. This function is used to determine which node to expand next based on each open node's closeness to the solution.

**DepthFirstSearch and BreadthFirstSearch Classes**

`Object[ ] solve (Solvable start, Solvable goal)`

Returns the solution path (in order, start is array element 0), from start to goal

Each search implemented in the API is represented by a single method, solve(). The solve() method's parameters are the starting state of the problem and the goal state of the problem. It returns an array of puzzle states that show each step on the way from start to solution. These two parameters passed to the solve() method must implement the Solvable interface.

**BestFirstSearch Class**

`Object[ ] solve (HSolvable start, HSolvable goal)`

Returns the solution path (in order, start is array element 0), using a best-first tree search, from start to goal

This class is very similar to the previously discussed search classes except that this solve() method requires that the parameters implement the Hsolvable interface. This is because the best-first search needs to know how to rank the different puzzle configurations in order of correctness. The Solvable interface does not provide for such a ranking. This rank is provided by the fitnessFunction() method in the HSolvable interface.

**AStarSearch Class**

`Object[ ] solve (HSolvable start, HSolvable goal)`

Returns the solution path (in order, start is array element 0), using the A* tree search, from start to goal

`Object[ ] solve (HSolvable start, HSolvable goal, int weight)`

Returns the solution path (in order, start is array element 0), using the A* tree search with weighting, from start to goal

The AStarSearch class is the only algorithm class with two different solve() methods. The first solve() method takes the same parameters as the solve() methods for the other heuristic algorithms. However, the second solve() method requires an additional parameter, an int. This int is multiplied by the depth of the node during the calculation of the heuristic value. The purpose of this second method is to provide the programmer with a way to modify the algorithm slightly to favor nodes that are logically closer to the solution over nodes that are higher up in the tree.

**ValuedNode Class (implements Comparable)**

`public ValuedNode()`

Creates a ValuedNode with value 32000 and null contents

`public ValuedNode(Object contents, int value)`

Creates a ValuedNode with specified contents and value

`public int compareTo(Object o)`

Returns a negative int, zero or positive int based on whether this ValuedNode is less than, equal to, or greater than the specified Object

`public Object getContents()`

Returns the contents of the Node

`public int getValue()`

Returns the value associated with the contents of the Node

This is the first of the support classes. The purpose of this class is to associate a value with each node in the open node list. Again, the open node list is used to keep track of nodes in the tree that are yet to be expanded. By implementing the Comparable interface,

a list of these objects can be easily sorted using the Collections.sort() method. The compareTo() function of this class allows the Collections.sort() method to sort a list of ValuedNode objects from smallest (at element 0) to largest using the value data member of the ValuedNode class.

**Tree Class**

`public Tree()`

Creates a Tree with null root

`public Tree(Object x)`

Creates a new Tree with root x

`public void addObject(Object newChild, Object parent)`

Adds object newNode to the Tree with parent as its parent

`public void addObjects(Object[] newNodes, Object parent)`

Adds objects newNodes to the Tree with parent as their parent

`public boolean contains(Object x)`

Returns true if x is present somewhere in the Tree using the equals() method for comparison

`public int getDepth(Object y)`

Returns the depth in the Tree of object y

`public Object getParent(Object y)`

Returns the parent object of object y

`public int getSize()`

Returns the total number of objects in the Tree

`public Object[] pathFromRoot(Object currentNode)`

Returns an array of all Nodes between the root and currentNode

`public Object[] pathToRoot(Object currentNode)`

Returns an array of all Nodes between the currentNode and the root

The Tree class is another support class. The Tree class provides the functionality for the algorithm classes to create a solution space and to move through the solution space. The programmer can add a single object or an array of objects to the Tree as long as the parent is

specified. The size of the Tree and the depth of any given object are also available through the API. Finally, the pathFromRoot() and pathToRoot() methods can be used to generate the return array for the algorithms once the solution is found.

There are other objects and methods defined in the code, but they are hidden from the programmer with the private keyword. These hidden objects and methods will be discussed in the implementation section when appropriate.

# Implementation

The implementation of the API consists of several classes. The code for each will be shown and then discussed in-depth. Because of the size of some of the classes, many will be discussed in sections.

## Solvable Interface

```
public interface Solvable
{
   Object[] generateChildren();

   boolean equals(Object o);
}
```

The Java code for interfaces is rather simple. This is because the methods do not need to be implemented. The code simply lists the descriptors for the methods required for a class to implement the interface. The code for each method is written in the class that implements the interface. Examples of how to implement these methods will be discussed in the next section on how to use the API.

## HSolvable Interface

```
public interface HSolvable extends Solvable
{
   int fitnessFunction(Object o);
}
```

The most important thing to note in this code is how this interface extends Solvable. This means that all of the methods (and data members) from Solvable are also contained in HSolvable. Therefore, an HSolvable object can be used in place of a Solvable object because HSolvable is a more specific version of a Solvable object. In other words, HSolvable is a sub-interface of Solvable.

## TreeSearch Class

This is the class that all brute force algorithms are descended from. It defines the general algorithm used to perform all tree searches. The TreeSearch class is abstract, which

means that it cannot be instantiated, only inherited by other classes. This is because the class defines one abstract method, chooseNode(). The descended algorithm classes must implement this method.

This implementation was chosen because all tree search algorithms follow a basic set of steps, and really only differ in how they choose the next node to expand. So, the TreeSearch class defines the solve() method and then calls an abstract method, chooseNode() to perform this decision.

```
public abstract class TreeSearch
{
    Vector openNodes;
    Tree solutionSpace;

   public TreeSearch()
    {
    }

    public Object[] solve(Solvable start, Solvable solution)
   {
        Solvable currentNode = start;
        openNodes = new Vector();
        solutionSpace = new Tree(currentNode);
```

This first section of code simply declares the class, the constructor, and the only method, solve(). It also declares and initializes all of the variables that are going to be needed. Notice that the list of open nodes is stored in a Vector. A Vector is simply a variable length array. Objects can be added and removed from it with the add() and remove() methods defined in the Vector class. A Vector was chosen over an array because some problems generated a large open node list. It is not memory efficient to define a large array for a problem that may need only a few spaces in openNodes. A Vector's space is dynamically allocated based on how many objects are currently in the list. This means small lists use a small space and large lists use a large space.

One other thing to notice is that the variable currentNode is initialized as an interface, not an object. In Java, it is acceptable to have a variable type that is defined by an

interface. This is done because the code does not know the kind of problem it is going to solve. It can only recognize that the problem was defined in a class that implements the Solvable interface. When an object is declared as an interface, only methods in the interface may be used with the object. Finally, a Tree with the start object at the root is created to represent the solution space.

```
if(start.equals(solution))
   return solutionSpace.pathFromRoot(currentNode);

while(true)
{
   Object[] children = currentNode.generateChildren();
```

Before the main algorithm loop even starts, the code checks to make sure that the solution and the start parameters are not the same. If they are not, the algorithm starts to go through the Tree looking for the solution. The first step is to expand currentNode. To accomplish this, the generateChildren() method is called on the currentNode and an array of objects is returned.

```
if(children != null)
{
   for(int i=0;i<children.length;i++)
   {
      if(children[i].equals(solution))
         solutionSpace.addObject(children[i], currentNode);
         return solutionSpace.pathFromRoot(children[i]);
      if(!solutionSpace.contains(children[i]))
      {
         solutionSpace.addObject(children[i], currentNode);
         openNodes.add(children[i]);
      }

   }
}
```

If the array of children is not null (if there are children for the current node), then each child is compared to the solution. If the solution is found, the path to the child from the root is returned. If the child is not the solution, then the code checks to see if it is in the solution space. If the child is unique, it is added to the Tree and openNodes.

```
            if(openNodes.size()==0)
                return null;
            else
                currentNode = (Solvable)this.chooseNode(openNodes);
        }
    }
```

After the children are added to the solution space, a new currentNode must be picked. If openNodes is empty, then all possible nodes have been examined and no solution is possible. Otherwise, if there are nodes in the open list, then one is chosen to be the next currentNode. The next node is selected differently depending on the algorithm. That is why the abstract method chooseNode() is called. Each class that inherits TreeSearch defines this method that returns the desired node.

```
    abstract public Object chooseNode(Vector x);
}
```

The last part of this abstract class defines the abstract method chooseNode(). It is defined abstract so that all inheriting classes are forced to define this method or be abstract themselves.

**DepthFirstSearch Class**

```
public class DepthFirstSearch extends TreeSearch
{
    public Object chooseNode(Vector x)
    {
        return x.remove(x.size()-1);
    }
}
```

By extending the TreeSearch class, the code required for the DepthFirstSearch class is very small. The chooseNode() method simply takes most recently added Node to the openNodes and returns it, making it currentNode. This is consistent with the algorithm as it was defined before because the most recently added node is always child of the current node. If the current node has no children, then the selection of the most recently added node forces the algorithm to backtrack up the tree just as it is supposed to.

**BreadthFirstSearch Class**

```
public class BreadthFirstSearch extends TreeSearch
{
   public Object chooseNode(Vector x)
   {
      return x.remove(0);
   }
}
```

The code for the BreadthFirstSearch is also very short because of its inheritance of TreeSearch. Instead of choosing the most recently added node like the DepthFirstSearch class, the BreadthFirstSearch class chooses the node that has been in the open list the longest. This forces this algorithm to search across the Tree, just like the breadth-first search is designed to do.

**HTreeSearch Class**

```
import java.util.Vector;

public abstract class HTreeSearch
{
   Vector openNodes;
   Tree solutionSpace;

   public HTreeSearch()
   {
   }

   public Object[] solve(HSolvable start, HSolvable solution)
   {
      HSolvable currentNode = start;
      openNodes = new Vector();
      solutionSpace = new Tree(currentNode);

      if(start.equals(solution))
         return solutionSpace.pathFromRoot(currentNode);

      while(true)
      {
         Object[] children = currentNode.generateChildren();

         if(children != null)
         {
            for(int i=0;i<children.length;i++)
            {
               if(children[i].equals(solution))
                  solutionSpace.addObject(children[i], currentNode);
                  return solutionSpace.pathFromRoot(children[i]);
               if(!solutionSpace.contains(children[i]))
               {
                  solutionSpace.addObject(children[i], currentNode);
                  openNodes.add(new ValuedNode(children[i],
                     HValue(children[i],solution)));
               }
```

```
            }
         }
         if(openNodes.size()==0)
            return null;
         else
            currentNode = (HSolvable)this.chooseNode(openNodes);
      }
   }

   abstract public Object chooseNode(Vector x);

   abstract public int HValue(Object x, HSolvable goal);
}
```

The HTreeSearch class is the heuristic version of the TreeSearch class. The heuristic algorithms inherit this class because it replaces all of the Solvable references in TreeSearch with HSolvable. It also defines an additional abstract method, Hvalue() which inheriting classes must implement. Although this value is usually based on the fitnessFunction() method, Hvalue() is defined loosely enough for the heuristic value to be just about anything. The integer returned by HValue is used to create a ValuedNode, which contains the child and the fitness value of the child. Without the ValuedNode, the open node list would not be able to be sorted.

**BestFirstSearch Class**

```
public class BestFirstSearch extends HTreeSearch
{
    public Object chooseNode(Vector x)
    {
      Collections.sort(x);
      return ((ValuedNode)openNodes.remove(0)).getContents();
    }

    public int HValue(Object x, HSolvable goal)
    {
      return ((HSolvable)x).fitnessFunction(goal);
    }
}
```

The BestFirstSearch class inherits HTreeSearch, so the class itself is rather short because HTreeSearch takes care of defining the general algorithm. The method chooseNode() first sorts the open node list. After the sort, the lowest valued (most correct) nodes are at the beginning of the Vector, and the first element in the Vector is chosen to be

the current node. The Hvalue() method simply returns the fitnessFunction() value of the node as specified in the best-first algorithm.

**AStarSearch Class**

```
public class AStarSearch extends HTreeSearch
{
   int weight = 1;

   public Object[] solve(HSolvable start, HSolvable solution, int x)
   {
      weight = x;
      return solve(start, solution);
   }

   public Object chooseNode(Vector x)
   {
      Collections.sort(x);
      return ((ValuedNode)openNodes.remove(0)).getContents();
   }

   public int HValue(Object x, HSolvable goal)
   {
      return ((HSolvable)x).fitnessFunction(goal) +
         solutionSpace.getDepth(x)*weight;
   }

}
```

The AStarSearch also inherits the HTreeSearch. AStarSearch also defines an additional data member, weight. This weight is used to calculate the return value of the Hvalue() function. The weight factor is included to let the programmer assign less importance to the depth of the node and more importance to the fitness of the node.

This class also defines an additional solve() method. It overloads the inherited solve() method with a method that takes two HSolvable objects and an int for parameters. When this solve() method is called, the weight is set equal to the int parameter, and then the original solve() defined in HTreeSearch is called.

The chooseNode() method here is identical to the one for the BestFirstSearch. The difference between the two is in the Hvalue() method. The AStarSearch adds the depth of the node onto the fitness function as the A* algorithm specifies.

**ValuedNode Class**

The purpose of this class is to provide a simple way to associate a value with an object and to sort those objects based on that value.

```
public class ValuedNode implements Comparable
{
   Object contents;
   int value;

   public ValuedNode()
   {
      contents = null;
      value = 32000;
   }

   public ValuedNode(Object x, int y)
   {
      contents = x;
      value = y;
   }
```

The ValuedNode class contains two data members: the contents of the node and the value associated with those contents. There are two constructors for the ValuedNode class. One is the null constructor, which creates a ValuedNode with no contents and a high value. The other constructor creates a ValuedNode with the specified value and contents.

```
   public int compareTo(Object o)
   {
      int value2 = ((ValuedNode)o).getValue();
      int diff = value - value2;
      return diff;
   }
```

This method implements the Comparable interface that allows the ValuedNode class to be sorted when in a list. This compareTo() method specifies that ValuedNodes be sorted based on their value data member from low value (at element 0) to high value. This could easily have been switched by swapping the variables value and value2 in the second line of the function.

```
    public Object getContents()
    {
       return contents;
    }

    public int getValue()
    {
       return value;
    }
}
```

These last two functions are simply access functions that allow the programmer to retrieve the value of the data members of a ValuedNode. Since there are no set functions, the data in a ValuedNode is read-only once the ValuedNode is created.

**Tree Class**

The Tree class could be considered the heart and soul of this API. It is the main data structure used by all of the algorithms and stores the solution space that is being searched. It is also the largest of all of the classes in terms of code size.

To understand the Tree class, the Node class must first be introduced. The Node class is a private class within the Tree class that links together each node of the Tree. A Java class is not normally defined within another class, but it was done in this case because a Tree is the only class that needs to use the Node class. The Node class is not a public class for another reason. This API was designed to be as simple to use as possible, and to this end, there is no need to trouble the programmer with the manipulation of Node objects. By hiding the Node class and letting the programmer work with Objects, the API creates a layer of abstraction and lets the programmer concentrate on more important programming matters.

```
class Node
{
   Object contents;
   Vector children;
   Node parent;

   Node()
   {
      contents = null;
      children = new Vector();
      parent = null;
   }

   Node(Object x)
   {
      this();
      contents = x;
   }
```

The Node class consists of three data members. The contents data member is the actual object that the node represents. The children Vector maintains a list of all of the children of the node. Finally, the parent data member contains a reference to the parent of the Node. This is a double-linked tree because each Node keeps track of its parent and its children. This allows a programmer to traverse a Tree upward or downward, not just one or the other. It takes more memory to link a tree in this manner, but gives more flexibility.

```
   public Object[] getChildren()
   {
      return children.toArray();
   }

   public Object getContents()
   {
      return contents;
   }

   public Node getParent()
   {
      return parent;
   }
```

These methods allow the Tree class to retrieve the data members of the Node class. Again, since the Node class is imbedded within the Tree class, only the Tree class can use these methods.

```
    public void addChild(Node x)
    {
        children.add(x);
    }

    public void setContents(Object x)
    {
        contents = x;
    }

    public void setParent(Node x)
    {
        parent = x;
    }
}
```

These methods allow the Tree class to set certain data members of the Node class. The addChild() method adds the specified Node to the list of children. You cannot remove a child from a Node's children Vector.

The Node class discussion enables an explanation of the Tree class. The Tree class contains a few methods that allow the programmer to add objects to the Tree, move through the Tree, and find paths in the Tree.

```
public class Tree
{
    Node root;
    Hashtable objectTable;
    int size;
    boolean customHash;

    public Tree()
    {
        root = null;
        objectTable = new Hashtable();
        size = 0;
        customHash = false;
    }

    public Tree(Object x)
    {
        this();

        Node newNode = new Node(x);

        objectTable.put(x, newNode);
        root = newNode;
        customHash = isCustomHash(x);
        size = 1;
    }
```

The Tree consists of two main data members, the root of the Tree and the objectTable. The objectTable is a Hashtable that performs the translation between the

Object that the programmer passes to a method and the Node that contains that Object. This is required because an Object itself does not contain information about its children or parent. The objectTable performs the translation between Object and Node without the knowledge of the programmer. When a programmer calls a Tree method, the Object parameters are converted to Nodes by the objectTable with the Hashtable's get() method. The objectTable contains all of the Objects (and Nodes) currently in the Tree. Therefore, the size of the Tree is kept track of with the size data member. The customHash data member keeps track of whether the objects in the Tree have a custom hashCode() method defined for them. The purpose of this will be obvious when the contains() method is discussed.

There are two constructors for the Tree class. One is the null constructor and simply creates a Tree with no root. The other constructor takes an Object as an argument and sets that Object as the Tree's root. The Object is placed in a Node by creating a new Node. The Object and the Node are then placed in the objectTable so that the Node can be looked up later if given the Object.

```
public void addObject(Object newChild, Object parent)
{
   Node newNode = new Node(newChild);
   addNode(newNode, (Node)objectTable.get(parent));
}

public void addObjects(Object[] newNodes, Object parent)
{
   for(int i = 0;i<newNodes.length;i++)
      addObject(newNodes[i], parent);
}

private void addNode(Node newNode, Node parent)
{
   objectTable.put(newNode.getContents(), newNode);
   size++;

   if(root==null)
   {
      root = newNode;
      customHash = isCustomHash(newNode.getContents());
   }
   else
   {
      newNode.setParent(parent);
      parent.addChild(newNode);
   }
}
```

These methods are used to add Objects to the Tree. Notice that only the first two methods are public. The private method, addNode(), is used by the first two public methods to actually perform the linking of the parent and child Objects. The method addObject() takes the Object parameter and wraps it in a Node. Secondly, it calls addNode() using the newly created Node as a parameter and looks up the parent Node given the parent Object in the objectTable. The method addObjects() simply calls addObject() once for each Object in the array.

The addNode() method is the method that actually places the Objects in the Tree. It first adds the Object and Node to the objectTable and increases the Tree size by one. Then, if there is no root, the method makes the Node the root and checks to see if a custom hashCode() function has been defined for the newly added Object. If there are already nodes in the Tree, the method sets the child Node's parent and adds the child Node to the parent's list of children.

```
public boolean contains(Object x)
{
   if(customHash)
      return objectTable.containsKey(x);
   else
   {
      Enumeration z = objectTable.keys();

      while(z.hasMoreElements())
      {
         if(x.equals(z.nextElement()))
            return true;
      }
      return false;
   }
}

private boolean isCustomHash(Object x)
{
   Method hashMethod = null;

   try{
      hashMethod = x.getClass().getMethod("hashCode",null);
   }catch(Exception z)
   {
      return false;
   }
   return hashMethod.getDeclaringClass().
      getName().equals(x.getClass().getName());
}
```

These two methods are responsible for searching the Tree to see if an Object is already in the Tree. The private method isCustomHash() uses the Java reflection API to determine if a method called hashCode() has been defined for the Objects within the Tree. It looks for a method called hashCode() and then checks to see if that method was inherited. The method returns true if a non-inherited hashCode() function exists.

The contains() method uses this information to determine how to perform the search. A linear search through the objectTable can take a long time, especially as the number of Objects increases. However, this is the only alternative if no hashCode() method is defined. Conversely, a defined hashCode() method enables the method containsKey() to quickly determine whether the Object is indeed in the objectTable.

The containsKey() method requires a custom hashCode() method because the inherited hashCode() function, which is defined in the Object class, uses a unique identifier, the Object ID, to calculate the hash code. As a result, two Objects with the same value for

all of its data members (that are equal by the equals method) may generate two different hash codes. Subsequently, the containsKey() function could return false even though two equal Objects are in the objectTable. The method containsKey() does not search the Hashtable completely, but generates the hash code for the requested Object and looks in that Hashtable slot. Even though the search is completed more quickly, the containsKey() method creates a possibility for error to an uninformed programmer.

The contains() method performs the linear search by default. However, if a custom hashCode() function exists, the quicker containsKey() method is used. The method performs the linear search by obtaining an Enumeration (ordered list) of all of the keys from the Hashtable and then searching through this list for the desired Object.

```
public int getDepth(Object y)
{
   int i = 0;
   Node x = (Node)objectTable.get(y);

   while(x != root)
   {
      i++;
      x = x.getParent();
   }

   return i;
}

public Object getParent(Object y)
{
   Node x = (Node)objectTable.get(y);
   Node parent = x.getParent();

   return parent.getContents();
}

public int getSize()
{
   return size;
}
```

These methods are used to obtain certain properties of the Tree or Nodes of the Tree. The getDepth() method returns the depth of an Object in the Tree. This method starts a counter at 0 and then performs the getParent() method until the root node is reached. The number of times getParent() is called is the depth of the node.

The getParent() method is simply a wrapper for the Node getParent() method. The Object is looked up in the objectTable and the retrieved Node is used to call the Node class' getParent() method. The contents of the returned Node are then returned to the method caller as an Object. The getSize() method returns the size of the Tree by examining the size data member.

```
   public Object[] pathFromRoot(Object currentNode)
   {
      Vector solutionPath = new Vector();

      solutionPath.add(currentNode);

      while(getDepth(currentNode) > 0)
      {
         currentNode = getParent(currentNode);
         solutionPath.insertElementAt(currentNode,0);
      }

      Object solutionArray[] = solutionPath.toArray();

      return solutionArray;
   }

   public Object[] pathToRoot(Object currentNode)
   {
      Vector solutionPath = new Vector();

      solutionPath.add(currentNode);

      while(getDepth(currentNode) > 0)
      {
         currentNode = getParent(currentNode);
         solutionPath.add(currentNode);
      }

      Object solutionArray[] = solutionPath.toArray();

      return solutionArray;
   }
}
```

These last two methods are used to return a path through the Tree. The algorithm classes use these methods to determine the path from the start (root) to the solution. These methods start at the solution and use the getParent() method until they arrive at the root. Each adds any Object encountered on the way to a Vector. The Vector is then converted to an array and returned. The pathToRoot() method adds each Object to the end of the Vector, and the pathFromRoot() method inserts each Object at the beginning of the Vector.

The two data structures, the openNodes Vector of the algorithm classes and the objectTable Hashtable of the Tree class might seem redundant. The option of using only one of these lists was considered. However, these similar structures maintain different information. The objectTable stores all of the configurations generated thus far. The openNodes Vector preserves a record of the nodes that have not yet been expanded. One represents the open list of nodes, and the other the closed list. Once expanded, a node is removed from the Vector. A flag could be added to each entry in the objectTable to maintain a list of expanded nodes. However, choosing the next node to expand would then become problematic. A Hashtable cannot be sorted, so there would be no simple way to find the unexpanded nodes in the Hashtable.

## Use of the API

The API can be used in many different ways. For instance, the API could be used to define another search object that implements a different tree search algorithm. The algorithms that are already defined in the API provide a good reference for this use. The programmer could either extend the TreeSearch or HTreeSearch class or use the Tree class to construct a solution space and write an algorithm to manipulate and search the Tree.

The API can also be used to find the solution to a puzzle and determine the efficiency of a particular algorithm. To use the API, a puzzle class must implement the HSolvable or Solvable interface. The programmer would then select a search to use and simply call the solve method of that search object. Once the start and end puzzle states are defined by the programmer, different search methods could be used to solve the problem. Below are two examples of how the API can be used to solve problems. A 8-puzzle demonstrates the API's decision-making capabilities. A maze displays the API's path finding ability.

### 8-Puzzle

The 8-puzzle (Figure 6) is a popular puzzle game that consists of a 3x3 grid with 8 numbered tiles that can be slid around the grid. The object is to get the tiles from one configuration to another. A common goal configuration is the one shown on the right. It has been implemented as a tree search problem before (Van Opdorp), but here this API will be used to make development easier.

| 1 | 2 | 3 |
|---|---|---|
| 4 | 5 | 6 |
| 7 | 8 | |

Figure 6: 8-Puzzle

To reiterate, the programmer begins by defining the Solvable or HSolvable interface. In both examples, the HSolvable interface is defined so that a variety of sample

fitnessFunction() methods can be discussed. As before, the code will be discussed in sections.

```
public class EightPuzzle implements HSolvable
{
   int[][] squares;

   EightPuzzle()
   {
      int[][] x = {{1,2,3},{4,5,6},{7,8,0}};
      squares = x;
   }

   EightPuzzle(int[][] x)
   {
      squares = x;
   }
```

In this section of code, the puzzle class is defined with the intention of implementing the HSolvable interface. The only data member required is an array to store the location of each numbered tile within the grid. If the first tile is at a grid location, the value of the grid at that location will be 1, the second tile will be represented by a value of 2, and so on. A zero represents the blank space in the grid.

The class has two constructors. If the null constructor is used, a default configuration is loaded. The second constructor allows the programmer to specify which array to use.

```
public Object clone()
{
   int y[][] = {{squares[0][0],squares[0][1],squares[0][2]},
             {squares[1][0],squares[1][1],squares[1][2]},
             {squares[2][0],squares[2][1],squares[2][2]}};

   return new EightPuzzle(y);
}

public int hashCode()
{
   int sum = 0;

   for(int i=0;i<3;i++)
      for(int j=0;j<3;j++)
         sum = sum + (i*3+j)*squares[i][j];

   return sum;
}
```

These two methods override methods defined in theObject class. The clone() method is defined so that the generateChildren() method can create more instances of the object. Although not part of either interface, the clone() method is usually necessary to correctly implement the generateChildren() method. This puzzle class also implements the hashCode() function to facilitate faster searching of the Tree as explained in the previous section.

```
public Object[] generateChildren()
{
   Vector children = new Vector();
   int i = 0,j = 0;

   while((j!=3)&&(squares[i][j] != 0))
   {
      i++;
      if(i==3)
      {
         i = 0;
         j++;
      }
   }
```

This is the first part of the generateChildren() method that is required to implement the HSolvable interface. This while loop searches the array for the blank grid space. Once the blank space is found, the code tries moving the empty space up, down, left or right. Although the tiles are usually moved in this puzzle, it is easier think of it as the blank space getting moved for coding purposes.

```
try
{
   if(j==3)
      return null;
   else
   {
      EightPuzzle copy = (EightPuzzle)this.clone();

      if(i!=0)    //move 0 left??
      {
         copy.setSquare(i,j,getSquare(i-1,j));
         copy.setSquare(i-1,j,0);
         children.add(copy);
      }
      if(i!=2)    //move 0 right??
      {
         copy = (EightPuzzle)this.clone();
         copy.setSquare(i,j,getSquare(i+1,j));
         copy.setSquare(i+1,j,0);
         children.add(copy);
      }
      if(j!=0)    //move 0 up??
      {
         copy = (EightPuzzle)this.clone();
         copy.setSquare(i,j,getSquare(i,j-1));
         copy.setSquare(i,j-1,0);
         children.add(copy);
      }
      if(j!=2)    //move 0 down??
      {
         copy = (EightPuzzle)this.clone();
         copy.setSquare(i,j,getSquare(i,j+1));
         copy.setSquare(i,j+1,0);
         children.add(copy);
      }
   }
} catch (Exception z)
{
   System.out.println(z.getMessage());
   return null;
}
```

The next step is moving the blank space. There are four main parts of this section, one for each direction. For each direction, the code examines the blank space's maneuverability to ensure that it is not on the edge of the grid. If the blank space can be moved, a clone is created and the tiles are moved on the clone. Finally, the clone is added to the children Vector.

Notice that this whole section is in a try block. This is because the clone() method can return a CloneNotSupported exception. This exception occurs if the Object clone() method cannot be used and it is not overwritten by a valid clone() method.

A clone() method is necessary because the movement of the tiles in the same object only results in the same configuration in the Vector four times. This is because a reference to the Object, not the contents of the Object itself, is stored in the Vector. Consequently, when the tiles are switched for one direction, the tiles are also switched in the Object that has already been added to the Vector.

```
        if(children.size()==0)
            return null;
        else
            return children.toArray();
    }
```

Finally, the method must convert the children Vector to an array of Objects and return the array. If there are no valid children, null is returned.

```
    public boolean equals(Object x)
    {
        boolean same = true;

        EightPuzzle y = (EightPuzzle)x;
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<3;j++)
            {
                if(y.getSquare(i,j) != squares[i][j])
                    same = false;
            }
        }
        return same;
    }
```

The equals() method is the second of three functions required by the HSolvable interface. This method overwrites the equals() method defined in Object. The equals() method scans through the grid array and returns false if any of the array elements do not match.

```
public int fitnessFunction(Object x)
{
    EightPuzzle solution = (EightPuzzle)x;
    int numWrong = 0;

    for(int i=0;i<3;i++)
    {
        for(int j=0;j<3;j++)
        {
            if(solution.getSquare(i,j)!=squares[i][j])
                numWrong++;
        }
    }
    return numWrong;
}
```

The most important method of the HSolvable interface is the fitnessFunction() method. A good fitnessFunction() will result in a quick solution. A poor one will result in a nearly random search of the solution space. The fitnessFunction() method returns an int that is representative of how close the puzzle configuration is to the passed puzzle configuration. A high int is representative of a large difference between the configurations and a small int represents the configurations being very similar. Although not usually the case, the fitnessFunction() method can return a negative integer. In negative valued fitness functions, a –1 is closer to the solution than a 0.

This sample fitness function progresses through the grid array and counts how many array elements are different. It gives a maximum return value of 9, and a minimum of 0. One limitation of this fitness function is the small range of values it can return. A good fitness function will have a large range of values with many different possible values in between the minimum and maximum. For instance, simply multiplying the fitness value does not make it a better fitness function. Good fitness functions have many different possible values over a large range.

```
   public int getSquare(int i, int j)
   {
      return squares[i][j];
   }

   public void setSquare(int i, int j, int value)
   {
      squares[i][j] = value;
   }

   public void printAsText()
   {
      System.out.println(" "+squares[0][0]+" "+squares[0][1]+
         " "+squares[0][2]);
      System.out.println(" "+squares[1][0]+" "+squares[1][1]+
         " "+squares[1][2]);
      System.out.println(" "+squares[2][0]+" "+squares[2][1]+
         " "+squares[2][2]);
   }
```

These functions are used to access the grid array and to print out an instance of the puzzle in a logical manner.

```
   public static void main(String args[])
       throws CloneNotSupportedException
   {
      int[][] y = {{1,4,3},{7,2,5},{0,8,6}};
      int[][] z = {{1,2,3},{4,5,6},{7,8,0}};

      EightPuzzle start = new EightPuzzle(y);
      EightPuzzle end = new EightPuzzle(z);
      BestFirstSearch search = new BestFirstSearch();

      Object[] solution = search.solve(start,end);

      if(solution == null)
         System.out.println("No Solution Possible");
      else
      {
         for(int i=0;i<solution.length;i++)
         {
            ((EightPuzzle)solution[i]).printAsText();
         }
      }
   }
}
```

The last part of this sample problem is the main() executable method. This method creates two instances of the puzzle representing the starting point and the end point. An instance of an algorithm class is then created and the solve() method called. An array of Objects is returned from the solve() method. This array is then printed out from start to finish so the user can see the steps from start to solution.

The 8-puzzle was solved using the four algorithms. Two different start and end configurations are shown below with the average time each algorithm took to solve the examples.

| | 4 | 2 |
|---|---|---|
| 1 | 7 | 3 |
| 8 | 6 | 5 |

Figure 7: Ex. 1 - Start

| 1 | 2 | 3 |
|---|---|---|
| 4 | 5 | 6 |
| 7 | 8 | |

Figure 8: Ex. 1 - Goal

Depth-first Search: >15 minutes
Best-first Search: 50ms – 110ms

Breadth-first Search: 600ms – 660ms
A* Search: 160ms – 220ms

| 1 | 2 | 3 |
|---|---|---|
| 4 | 5 | 6 |
| 7 | 8 | |

Figure 9: Ex. 2 - Start

| 1 | 2 | 3 |
|---|---|---|
| 8 | | 7 |
| 6 | 5 | 4 |

Figure 10: Ex. 2 - Goal

Depth-first Search: 425ms
Best-first Search: 1700ms

Breadth-first Search: 95000ms
A* Search: 2450ms

From the results of these two examples, it is obvious that there is not one single algorithm that works the best for all cases of the 8-puzzle. It must be noted that certain algorithms function better in different circumstances even within the same puzzle. For instance, the second example is specifically designed to work well with the depth-first algorithm. The heuristic algorithms perform more consistently in these examples. This holds true in most cases because they actively search out the solution. The A* algorithm probably turns in slightly slower solution times than the best-first here because the A* also tries to find the shortest path. This added effort can cause the algorithm to take longer, but can sometimes yield a better solution.

**Maze Puzzle**

The maze puzzle is a much simpler problem to solve than the 8-puzzle. This example maze consists of a square grid measuring 14 squares across and 7 squares down. The maze is shown to the right. The upper left corner has been designated (0,0), with the first coordinate pertaining to the column and the second to the row. The maze puzzle will be explained in a similar fashion to the 8-puzzle.

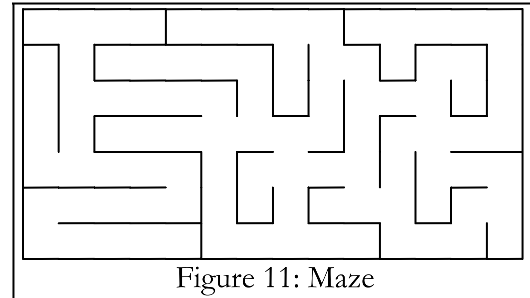
Figure 11: Maze

```
public class TestMaze implements HSolvable
{
   int[][] squares = {{11,2,10,14,11,10,10,2,6,11,2,10,10,6},
                      {7,5,11,10,10,10,6,5,1,6,13,3,6,5},
                      {5,1,10,10,10,6,5,13,5,1,10,4,13,5},
                      {5,5,11,10,10,0,8,2,12,5,3,0,10,12},
                      {9,8,10,10,6,5,3,0,10,4,5,1,10,6},
                      {3,10,10,10,12,5,13,5,11,10,4,13,3,4},
                      {9,10,10,10,14,9,10,8,10,14,9,10,12,13}};
   Point location;
   static final int LWALL=1,UWALL=2,RWALL=4,DWALL=8;

   TestMaze()
   {
      location = new Point(0,0);
   }

   TestMaze(Point x)
   {
      location = x;
   }
```

The TestMaze class implements the HSolvable interface in order to use the heuristic algorithm classes, BestFirstSearch and AStarSearch. The class consists of an array that stores the configuration of the walls of each grid space and a Point that holds the current position of the marker in the maze. The wall configuration of each grid square is stored by assigning a bit to each of the four walls. The left wall is worth 1, the top wall is worth 2, the right wall is worth 4 and the bottom wall is worth 8. The value of an array element is calculated by adding the values of all walls present at the corresponding grid coordinate. The two

constructors allow the programmer to either specify the location of the marker or to let the default value (0,0) be applied.

```
public Object clone()
{
   return new TestMaze(new Point(location));
}

public int hashCode()
{
   return (int)(location.getX()*20 + location.getY());
}
```

These functions serve the same purpose here as in the 8-puzzle. Again, hashCode() is defined so that the tree search is performed faster.

```
public Object[] generateChildren()
{
   Vector children = new Vector();

   int squareVal =
      squares[(int)location.getY()][(int)location.getX()];

   try
   {
      if((squareVal & LWALL) == 0)     //move left??
      {
         TestMaze copy = (TestMaze)this.clone();
         copy.location = new Point((int)location.getX() –
            1,(int)location.getY());
         children.add(copy);
      }
      if((squareVal & UWALL) == 0)     //move up??
      {
         TestMaze copy = (TestMaze)this.clone();
         copy.location = new Point((int)location.getX(),
            (int)location.getY() - 1);
         children.add(copy);
      }
      if((squareVal & RWALL) == 0)     //move right??
      {
         TestMaze copy = (TestMaze)this.clone();
         copy.location = new Point((int)location.getX() +
            1,(int)location.getY());
         children.add(copy);
      }
      if((squareVal & DWALL) == 0)     //move down??
      {
         TestMaze copy = (TestMaze)this.clone();
         copy.location = new Point((int)location.getX(),
            (int)location.getY() + 1);
         children.add(copy);
      }
   } catch (Exception z)
   {
      System.out.print("Exception....");
      System.out.println(z.getMessage());
      return null;
   }
   if(children.size()==0)
```

```
            return null;
        else
            return children.toArray();
    }
```

The generateChildren() function for the maze is very similar to that of the 8-puzzle. There are four sections to the code here, one for each direction. For each if statement, the code tests for the presence of a wall in that direction. If not, a child is generated using the clone() method and the child is added to the child Vector. After the four directions are tested, the Vector is converted into an array of Objects and returned. Again, if there are no children, a value of null is returned.

```
    public boolean equals(Object x)
    {
        return location.equals(((TestMaze)x).getLocation());
    }
```

The equals() function for the maze is very simple. Since equals() is already defined for Point, this method is extremely trivial. The array does not need to be compared because it is defined in the class and unchangeable.

```
    public int fitnessFunction(Object x)
    {
        TestMaze solution = (TestMaze)x;
        int solutionX = (int)solution.getLocation().getX();
        int solutionY = (int)solution.getLocation().getY();

        int distanceX = (int)(location.getX() - solutionX);
        int distanceY = (int)(location.getY() - solutionY);

        int value = distanceX + distanceY;

        return value;
    }
```

The fitnessFunction() method for the maze is much better than the one for the 8-puzzle. This method determines the taxicab distance between the location data member of the two mazes. The maximum value is 14+7=21 and the minimum value is 0.

```
public Point getLocation()
   {
      return location;
   }

   public void setLocation(Point x)
   {
      location = x;
   }

   public void printAsText()
   {
      System.out.print(" ("+location.getX()+", "+location.getY()+") ");
   }
```

These methods provide access to the location data member and print out the location in a coordinate format.

```
   public static void main(String args[])
      throws CloneNotSupportedException
   {
      TestMaze start = new TestMaze(new Point(0,0));
      TestMaze end = new TestMaze(new Point(13,6));
      DepthFirstSearch search = new DepthFirstSearch();

      Object[] solution = search.solve(start,end);

      if(solution == null)
         System.out.println("No Solution Possible");
      else
      {
         for(int i=0;i<solution.length;i++)
         {
            ((TestMaze)solution[i]).printAsText();
         }
      }
   }
}
```

This example uses the DepthFirstSearch algorithm. Notice that even though this class implements HSolvable, DepthFirstSearch and BreadthFirstSearch can still be used because HSolvable is a sub-interface of Solvable. The brute force algorithms are generally slower than the heuristic algorithms, but they can be good for comparison purposes.

This program was run using all four algorithms. The starting point was the upper left of the grid, and the goal was the lower right. The solutions generated by each algorithm and the solution time is given below.

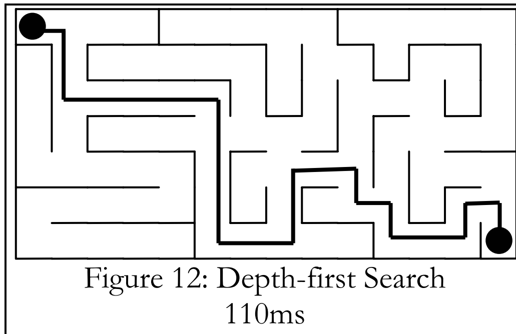
Figure 12: Depth-first Search
110ms

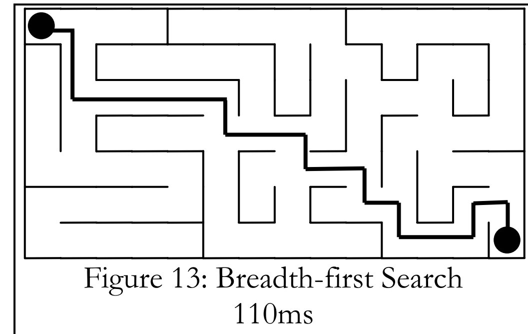
Figure 13: Breadth-first Search
110ms

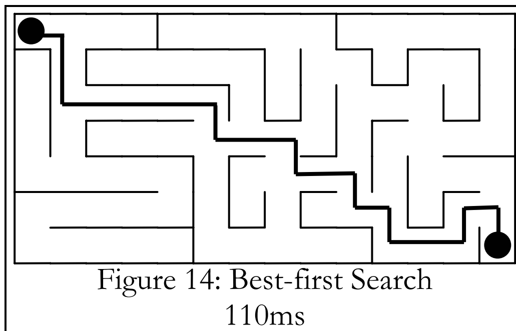
Figure 14: Best-first Search
110ms

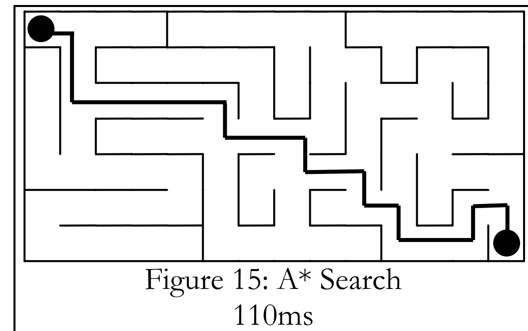
Figure 15: A* Search
110ms

These results show that for a small, simple problem, the algorithms are very similar. The only algorithm that returned a different path was the depth-first search and this path was only slightly longer than that returned by the others. All algorithms completed in the same amount of time due to the fact that this is a simple problem to solve. Even though the heuristic algorithms probably made fewer comparisons, their overhead to maintain a sorted list negated any gain fewer comparisons might have yielded.

**Other Applications**

This API has many other practical applications. One use for this API is education. This API could be a good learning tool for students. Using the four built-in algorithms, they can easily compare the solutions of different algorithms under many conditions. Students would describe the problem in terms of the HSolvable interface and the check the system time before and after the solve method to determine the total search time. Challenging

students to expand the API would also be a good project in a computer science or computer engineering course.

Another use for the API is Java-based game development. Simple AI algorithms are used often in gaming for computations, such as determining the next move in games like chess or checkers. These algorithms are also used for character path finding. Path finding is the ability of AI characters find their way across a map or a through a maze. Although Java is not used for complex three-dimensional graphics because of speed issues, many websites have multiplayer Java-enabled games. Instead of writing customized AI code for different games, a programmer could simply describe the game in terms of the Solvable interface and then use this API for computer players. As graphic accelerators become more powerful and move the graphic processing load off of the CPU, game developers have been able to add more complex physics and AI to games. This API can make the addition of these features easier.

## Conclusion

The API described above combines the research areas of tree search algorithms and object oriented programming to create a powerful and versatile tool. The API has no restrictions on the type of problem as long as it implements one of the interfaces. This allows the programmer to save time by not writing the algorithm code. The API enables this without introducing any restrictions on the problem to be solved.

One way to extend the usefulness of the API would be to reduce the interface requirement. This would still require the programmer to provide the API a set of rules about the problem, but could reduce the interface to one method, a validMove() method. This method would take two configurations of the puzzle and return true if the move was valid and false if not. The Java reflection API would be used to discover the fields of the puzzle class. This API could then adjust the field values and call the validMove() method to determine whether that adjustment was legal.

Another option would be to remove the interface all together and have the API discover 'move' methods of the puzzle. The programmer could specify all of the possible ways that a puzzle could change with methods and precede each of these method names with the word move. For instance, the 8-puzzle would have the methods moveTileUp(), moveTileDown(), moveTileLeft() and moveTileRight(). These methods would probably have to return a Boolean as well so that the API knows whether that move was valid in that case. The API could then discover these methods and use them, much like the JavaBeans architecture discovers properties and events.

The shear amount of documentation, written and electronic, on heuristic tree searches demonstrates a need for a quick way to implement tree search algorithms. This API could be used for any of the various heuristic search demonstration web sites

mentioned before. It also provides a starting point for an even more powerful framework that could use brand new ideas in AI problem solving such as intelligent rule discovery. The framework presented in this paper combines tree search algorithms with object-oriented technology to provide researchers and developers with a tool that can save time and reduce code complexity.

## References

Noonan, Robert E. *An Object-Oriented View of Backtracking*. Williamsburg, PA: College of William and Mary, 2000.

Tracy, Kim W. and Peter Bouthoorn. *Object-Oriented Artificial Intelligence Using C++*. New York: Computer Science Press, 1997.

Van Opdorp, Geert-Jan. 8 puzzle. http://www.aie.nl/~geert/java/public/EightPuzzle.html

Watson, Mark. Artificial Intelligence Search Techniques in Java. http://www.markwatson.com/opencontent/aisearch

## Bibliography

Bolc, Leonard and Jerzy Cytowski. *Search Methods for Artificial Intelligence*. London: Academic Press, 1992.

Java Game Programming Resources and Source Code. Altnet Inc. http://www.altnetinc.com/training/gdc99/source.htm

Nilsson, Nils J. *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.

Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley, 1984.

Tzeng, Chun-Hung. *A Theory of Heuristic Information in Game-Tree Search*. Berlin: Springer-Verlag, 1988.

# Appendix A – Source Code

**Solvable.java**

```
public interface Solvable
{
   Object[] generateChildren();

   boolean equals(Object o);
}
```

**HSolvable.java**

```
public interface HSolvable extends Solvable
{
   int fitnessFunction(Object o);
}
```

**ValuedNode.java**

```
import java.lang.Comparable;

public class ValuedNode implements Comparable
{
   Object contents;
   int value;

   public ValuedNode()
   {
      contents = null;
      value = 32000;
   }

   public ValuedNode(Object x, int y)
   {
      contents = x;
      value = y;
   }

   public int compareTo(Object o)
   {
      int value2 = ((ValuedNode)o).getValue();
      int diff = value - value2;
      return diff;
   }

   public Object getContents()
   {
      return contents;
   }

   public int getValue()
   {
      return value;
   }
}
```

**Tree.java**

```
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;
import java.lang.reflect.Method;

public class Tree
{
   Node root;
   Hashtable objectTable;
   int size;
   boolean customHash;

   public Tree()
   {
      root = null;
      objectTable = new Hashtable();
      size = 0;
      customHash = false;
   }

   public Tree(Object x)
   {
      this();

      Node newNode = new Node(x);

      objectTable.put(x, newNode);
      root = newNode;
      customHash = isCustomHash(x);
      size++;
   }

   public void addObject(Object newChild, Object parent)
   {
      Node newNode = new Node(newChild);
      addNode(newNode, (Node)objectTable.get(parent));
   }

   public void addObjects(Object[] newNodes, Object parent)
   {
      for(int i = 0;i<newNodes.length;i++)
         addObject(newNodes[i], parent);
   }

   public boolean contains(Object x)
   {
      if(customHash)
         return objectTable.containsKey(x);
      else
      {
         Enumeration z = objectTable.keys();

         while(z.hasMoreElements())
         {
            if(x.equals(z.nextElement()))
               return true;
         }
         return false;
      }
   }

   public int getDepth(Object y)
```

```
{
   int i = 0;
   Node x = (Node)objectTable.get(y);

   while(x != root)
   {
      i++;
      x = x.getParent();
   }

   return i;
}

public Object getParent(Object y)
{
   Node x = (Node)objectTable.get(y);
   Node parent = x.getParent();

   return parent.getContents();
}

public int getSize()
{
   return size;
}

public Object[] pathFromRoot(Object currentNode)
{
   Vector solutionPath = new Vector();

   solutionPath.add(currentNode);

   while(getDepth(currentNode) > 0)
   {
      currentNode = getParent(currentNode);
      solutionPath.insertElementAt(currentNode,0);
   }

   Object solutionArray[] = solutionPath.toArray();

   return solutionArray;
}

public Object[] pathToRoot(Object currentNode)
{
   Vector solutionPath = new Vector();

   solutionPath.add(currentNode);

   while(getDepth(currentNode) > 0)
   {
      currentNode = getParent(currentNode);
      solutionPath.add(currentNode);
   }

   Object solutionArray[] = solutionPath.toArray();

   return solutionArray;
}

private void addNode(Node newNode, Node parent)
{
   objectTable.put(newNode.getContents(), newNode);
   size++;
```

```
        if(root==null)
        {
            root = newNode;
            customHash = isCustomHash(newNode.getContents());
        }
        else
        {
            newNode.setParent(parent);
            parent.addChild(newNode);
        }
    }

    private boolean isCustomHash(Object x)
    {
        Method hashMethod = null;

        try{
            hashMethod = x.getClass().getMethod("hashCode",null);
        }catch(Exception z)
        {
            return false;
        }
        return hashMethod.getDeclaringClass().getName().
            equals(x.getClass().getName());
    }

    class Node
    {
        Object contents;
        Vector children;
        Node parent;

        Node()
        {
            contents = null;
            children = new Vector();
            parent = null;
        }

        Node(Object x)
        {
            this();
            contents = x;
        }

        public void addChild(Node x)
        {
            children.add(x);
        }

        public Object[] getChildren()
        {
            return children.toArray();
        }

        public Object getContents()
        {
            return contents;
        }

        public Node getParent()
        {
            return parent;
```

```
        }

        public void setContents(Object x)
        {
            contents = x;
        }

        public void setParent(Node x)
        {
            parent = x;
        }
    }
}
```

**TreeSearch.java**

```
import java.util.Vector;

public abstract class TreeSearch
{
    Vector openNodes;
    Tree solutionSpace;

   public TreeSearch()
    {
    }

    public Object[] solve(Solvable start, Solvable solution)
   {
        Solvable currentNode = start;
        openNodes = new Vector();
        solutionSpace = new Tree(currentNode);

        if(start.equals(solution))
         return solutionSpace.pathFromRoot(currentNode);

      while(true)
      {
         Object[] children = currentNode.generateChildren();

         if(children != null)
         {
            for(int i=0;i<children.length;i++)
            {
               if(children[i].equals(solution))
               {
                     solutionSpace.addObject(children[i], currentNode);
                  return solutionSpace.pathFromRoot(children[i]);
               }
               if(!solutionSpace.contains(children[i]))
               {
                     solutionSpace.addObject(children[i], currentNode);
                  openNodes.add(children[i]);
               }

            }
         }
         if(openNodes.size()==0)
            return null;
         else
            currentNode = (Solvable)this.chooseNode(openNodes);
      }
   }

   abstract public Object chooseNode(Vector x);
}
```

**HTreeSearch.java**

```
import java.util.Vector;

public abstract class HTreeSearch
{
    Vector openNodes;
    Tree solutionSpace;

   public HTreeSearch()
    {
    }

    public Object[] solve(HSolvable start, HSolvable solution)
   {
        HSolvable currentNode = start;
        openNodes = new Vector();
        solutionSpace = new Tree(currentNode);

        if(start.equals(solution))
         return solutionSpace.pathFromRoot(currentNode);

      while(true)
      {
         Object[] children = currentNode.generateChildren();

         if(children != null)
         {
            for(int i=0;i<children.length;i++)
            {
               if(children[i].equals(solution))
               {
                  solutionSpace.addObject(children[i], currentNode);
                  return solutionSpace.pathFromRoot(children[i]);
               }
               if(!solutionSpace.contains(children[i]))
               {
                  solutionSpace.addObject(children[i], currentNode);
                  openNodes.add(new ValuedNode(children[i],
                     HValue(children[i],solution)));
               }
            }
         }
         if(openNodes.size()==0)
            return null;
         else
            currentNode = (HSolvable)this.chooseNode(openNodes);
      }
   }

   abstract public Object chooseNode(Vector x);

   abstract public int HValue(Object x, HSolvable goal);
}
```

**DepthFirstSearch.java**

```
import java.util.Vector;

public class DepthFirstSearch extends TreeSearch
{
   public Object chooseNode(Vector x)
   {
      return x.remove(x.size()-1);
   }
}
```

**BreadthFirstSearch.java**

```
import java.util.Vector;

public class BreadthFirstSearch extends TreeSearch
{
   public Object chooseNode(Vector x)
   {
      return x.remove(0);
   }
}
```

**BestFirstSearch.java**

```
import java.util.Vector;
import java.util.Collections;

public class BestFirstSearch extends HTreeSearch
{
    public Object chooseNode(Vector x)
    {
      Collections.sort(x);
      return ((ValuedNode)openNodes.remove(0)).getContents();
    }

    public int HValue(Object x, HSolvable goal)
    {
      return ((HSolvable)x).fitnessFunction(goal);
    }
}
```

**AStarSearch.java**

```
import java.util.Collections;

public class AStarSearch extends HTreeSearch
{
   int weight = 1;

   public Object[] solve(HSolvable start, HSolvable solution, int x)
   {
      weight = x;
      return solve(start, solution);
   }

   public Object chooseNode(Vector x)
   {
      Collections.sort(x);
         return ((ValuedNode)openNodes.remove(0)).getContents();
   }

   public int HValue(Object x, HSolvable goal)
   {
      return ((HSolvable)x).fitnessFunction(goal) +
         solutionSpace.getDepth(x)*weight;
   }

}
```

**EightPuzzle.java**

```
import java.util.Vector;
import java.util.Hashtable;

public class EightPuzzle implements HSolvable
{
   int[][] squares;

   EightPuzzle()
   {
      int[][] x = {{1,2,3},{4,5,6},{7,8,0}};
      squares = x;
   }

   EightPuzzle(int[][] x)
   {
      squares = x;
   }

   public Object clone()
   {
      int y[][] = {{squares[0][0],squares[0][1],squares[0][2]},
                   {squares[1][0],squares[1][1],squares[1][2]},
                   {squares[2][0],squares[2][1],squares[2][2]}};

      return new EightPuzzle(y);
   }

   public int hashCode()
   {
      int sum = 0;

      for(int i=0;i<3;i++)
         for(int j=0;j<3;j++)
            sum = sum + (i*3+j)*squares[i][j];

      return sum;
   }

   public Object[] generateChildren()
   {
      Vector children = new Vector();
      int i = 0,j = 0;

      while((j!=3)&&(squares[i][j] != 0))
      {
         i++;
         if(i==3)
         {
            i = 0;
            j++;
         }
      }

      try
      {
         if(j==3)
            return null;
         else
         {
            EightPuzzle copy = (EightPuzzle)this.clone();

            if(i!=0)    //move 0 left??
```

```
            {
               copy.setSquare(i,j,getSquare(i-1,j));
               copy.setSquare(i-1,j,0);
               children.add(copy);
            }
            if(i!=2)    //move 0 right??
            {
               copy = (EightPuzzle)this.clone();
               copy.setSquare(i,j,getSquare(i+1,j));
               copy.setSquare(i+1,j,0);
               children.add(copy);
            }
            if(j!=0)    //move 0 up??
            {
               copy = (EightPuzzle)this.clone();
               copy.setSquare(i,j,getSquare(i,j-1));
               copy.setSquare(i,j-1,0);
               children.add(copy);
            }
            if(j!=2)    //move 0 down??
            {
               copy = (EightPuzzle)this.clone();
               copy.setSquare(i,j,getSquare(i,j+1));
               copy.setSquare(i,j+1,0);
               children.add(copy);
            }
         }
      } catch (Exception z)
      {
         System.out.println(z.getMessage());
         return null;
      }
      if(children.size()==0)
         return null;
      else
         return children.toArray();
   }

   public boolean equals(Object x)
   {
      boolean same = true;

      EightPuzzle y = (EightPuzzle)x;
      for(int i=0;i<3;i++)
      {
         for(int j=0;j<3;j++)
         {
            if(y.getSquare(i,j) != squares[i][j])
               same = false;
         }
      }
      return same;
   }

   public int fitnessFunction(Object x)
   {
      EightPuzzle solution = (EightPuzzle)x;
      int numWrong = 0;

      for(int i=0;i<3;i++)
      {
         for(int j=0;j<3;j++)
         {
            if(solution.getSquare(i,j)!=squares[i][j])
```

```
                numWrong++;
            }
        }

        return numWrong;
    }

    public int getSquare(int i, int j)
    {
        return squares[i][j];
    }

    public void setSquare(int i, int j, int value)
    {
        squares[i][j] = value;
    }

    public void printAsText()
    {
        System.out.println(" "+squares[0][0]+" "+squares[0][1]+"
            "+squares[0][2]);
        System.out.println(" "+squares[1][0]+" "+squares[1][1]+"
            "+squares[1][2]);
        System.out.println(" "+squares[2][0]+" "+squares[2][1]+"
            "+squares[2][2]);
    }

    public static void main(String args[]) throws
        CloneNotSupportedException
    {
        int[][] y = {{1,4,3},{7,2,5},{0,8,6}};
        int[][] z = {{1,2,3},{4,5,6},{7,8,0}};

        EightPuzzle start = new EightPuzzle(y);
        EightPuzzle end = new EightPuzzle(z);
        BestFirstSearch search = new BestFirstSearch();

        Object[] solution = search.solve(start,end);

        if(solution == null)
            System.out.print("No Solution Possible");
        else
        {
            for(int i=0;i<solution.length;i++)
            {
                ((EightPuzzle)solution[i]).printAsText();
            }
        }
    }
}
```

**TestMaze.java**

```
import java.util.Vector;
import java.awt.Point;

public class TestMaze implements HSolvable
{
   int[][] squares = {{11,2,10,14,11,10,10,2,6,11,2,10,10,6},
                      {7,5,11,10,10,10,6,5,1,6,13,3,6,5},
                      {5,1,10,10,10,6,5,13,5,1,10,4,13,5},
                      {5,5,11,10,10,0,8,2,12,5,3,0,10,12},
                      {9,8,10,10,6,5,3,0,10,4,5,1,10,6},
                      {3,10,10,10,12,5,13,5,11,10,4,13,3,4},
                      {9,10,10,10,14,9,10,8,10,14,9,10,12,13}};
   Point location;
   static final int LWALL=1,UWALL=2,RWALL=4,DWALL=8;

   TestMaze()
   {
      location = new Point(0,0);
   }

   TestMaze(Point x)
   {
      location = x;
   }

   public Object clone()
   {
      return new TestMaze(new Point(location));
   }

   public int hashCode()
   {
      return (int)(location.getX()*20 + location.getY());
   }

   public Object[] generateChildren()
   {
      Vector children = new Vector();

      int squareVal =
         squares[(int)location.getY()][(int)location.getX()];

      try
      {
         if((squareVal & LWALL) == 0)     //move left??
         {
            TestMaze copy = (TestMaze)this.clone();
            copy.location = new
               Point((int)location.getX() - 1,(int)location.getY());
            children.add(copy);
         }
         if((squareVal & UWALL) == 0)     //move up??
         {
            TestMaze copy = (TestMaze)this.clone();
            copy.location = new
               Point((int)location.getX(),(int)location.getY() - 1);
            children.add(copy);
         }
         if((squareVal & RWALL) == 0)     //move right??
         {
            TestMaze copy = (TestMaze)this.clone();
            copy.location = new
```

```
            Point((int)location.getX() + 1,(int)location.getY());
         children.add(copy);
      }
      if((squareVal & DWALL) == 0)     //move down??
      {
         TestMaze copy = (TestMaze)this.clone();
         copy.location = new
            Point((int)location.getX(),(int)location.getY() + 1);
         children.add(copy);
      }
   } catch (Exception z)
   {
      System.out.print("Exception....");
      System.out.println(z.getMessage());
      return null;
   }
   if(children.size()==0)
      return null;
   else
      return children.toArray();
}

public boolean equals(Object x)
{
   return location.equals(((TestMaze)x).getLocation());
}

public int fitnessFunction(Object x)
{
   TestMaze solution = (TestMaze)x;
   int solutionX = (int)solution.getLocation().getX();
   int solutionY = (int)solution.getLocation().getY();

   int distanceX = (int)(location.getX() - solutionX);
   int distanceY = (int)(location.getY() - solutionY);

   int value = distanceX + distanceY;

   return value;
}

public Point getLocation()
{
   return location;
}

public void setLocation(Point x)
{
   location = x;
}

public void printAsText()
{
   System.out.print("("+location.getX()+", "+location.getY()+") ");
}

public static void main(String args[]) throws
   CloneNotSupportedException
{
   DepthFirstSearch search = new DepthFirstSearch();
   System.out.println("");

   TestMaze start = new TestMaze(new Point(0,0));
   TestMaze end = new TestMaze(new Point(13,6));
```

```
        Object[] solution = search.solve(start,end);

        if(solution == null)
           System.out.print("No Solution Possible");
        else
        {
           for(int i=0;i<solution.length;i++)
           {
              ((TestMaze)solution[i]).printAsText();
           }
        }
    }
}
```

Marquette University

This is to certify that we have examined
this copy of the
Master's thesis by

Patrick J. Clemins, B.S.

and have found that it is complete
and satisfactory in all respects.

The thesis has been approved by:

____________________________________

Thesis Director, Department of Electrical and Computer Engineering

____________________________________

Dr. Mark Barnard

____________________________________

Dr. Xin Feng

Approved on

_______________________